Universidad Autónoma de Madrid

Escuela Politécnica Superior

Grado en Ingeniería Informática

Trabajo de Fin de Grado

Acceleration of Big Data/Hadoop applications using GPU's

Lennert Verboven
lennert.verboven@estudiante.uam.es

03/06/2015

Acceleration of Big Data/Hadoop applications using GPU's

Author: Lennert Verboven
Tutor: Iván González Martínez

High Performance Computing and Networking Research Group
Tecnología Electrónica y de las Comunicaciones
Escuela Politécnica Superior
Universidad Autónoma de Madrid
03/06/2015

Abstract

Abstract — The goal of this work is to see which differences graphical processing units can make while working with big data. Creating a real world application is not part of this goal. Data mining and big data analysis allows organizations to find useful information and insights from huge amounts of structured or unstructured data. This information can be used to improve all kinds of services to customers or get the upper hand over competitors. In the Information Age data mining is becoming more and more popular due to the fact that more data is available. This is where Hadoop comes in. Hadoop provides a scalable and free framework which enables data analysis and operations on big data. Hadoop is very innovative in the field of big data. GPU's are highly parallel processors. By executing an algorithm in parallel which would otherwise be executed sequentially an increase in performance can be obtained. The Hadoop framework will be combined with NVIDIA's CUDA to complete the task of data analysis. Hadoop will call CUDA from its mapper or reducer. The computational intensive task will be moved away from Java to CUDA and be done in parallel. This will be done by porting two algorithms to CUDA and run these on all the compute nodes of the Hadoop cluster. The two algorithms are *sum of first n integers* and *factorial n*. Both of them will be written in Java and CUDA in order to compare the execution time. Both implementations of the two algorithms will be executed for millions of different values for n. The CUDA implementations will compute multiple factorials or sums at the same time while the Java implementations will compute them all sequentially. This work will show that not every algorithm is a viable option. The algorithm has to match certain criteria in order to create an effective CUDA port. Being time consuming without the need for large amounts of input or output data is the most important criteria. Obviously the algorithm has to allow a parallel version. If the algorithm needs the result from the previous input value parallelizing it will be a lot harder. This is shown by the *factorial n* algorithm. This algorithm is computationally intensive but produces a lot of output data. Copying data to and from the GPU is a time consuming task. When a lot of input data is required or a lot of output data is generated, copying this data might have such an impact that there is no more gain in execution time when using CUDA. This work proves that certain algorithms which handle big data can be ported to CUDA with a decrease in execution time as result, while also showing that others do not meet the necessary requirements.

Key words — Hadoop, CUDA, JCuda, Java, C

Resumen

Resumen — El objetivo de este trabajo es para identificar las ventajas que ofrece el uso de unidades de procesamientos gráfico cuando se trabaja con grandes volúmenes de datos. La minería de datos y análisis big data permite a las organizaciones encontrar información útil de las enormes cantidades de datos estructurados y no estructurados. Esta información puede utilizarse para mejorar todo tipo de servicios a los clientes o conseguir ventaja sobre sus competidores. En la Era de la Información la minería de datos es cada vez mas popular debido que hay más datos disponibles. Aqui es donde entra Hadoop. Hadoop ofrece un conjunto de herramientas escalables y libres que permiten el análisis de datos y las operaciones con grandes cantidades de datos. Hadoop es muy innovador en el campo de big data. Los GPU's son procesadores altamente paralelos. Mediante la ejecución de un algoritmo en paralelo se puede mejorar el rendimiento del mismo algoritmo ejecutado secuencialmente. El framework Hadoop se combinará con CUDA de NVIDIA para completar la tarea de análisis de datos. Hadoop llamará CUDA de su mapper o reducer. La tarea intensiva computacional se alejó de Java para CUDA y hacerse en paralelo. La integración de Hadoop y CUDA se ha realizado siendo posible ejecutar algoritmos en los nodos de cómputo del clúster Hadoop con GPUs. Los dos algoritmos son la *suma de los primeros n enteros* y *factorial n*. Ambos serán escritos en Java y CUDA con el fin de comparar el tiempo de ejecución. Ambas implementaciones de los dos algoritmos se ejecutarán para millones de diferentes valores de n. Las implementaciones CUDA computarán múltiples factoriales o sumas, al mismo tiempo, mientras que las implementaciones Java computarán todos ellos secuencialmente. Este trabajo demuestra que no todos los algoritmos son opciónes viables. El algoritmo tiene que cumplir con ciertos criterios para poder portarlos a CUDA y obtener buenos rendimientos. Si consumen mucho tiempo de ejecución sin la necesidad de grandes cantidades de datos de entrada o de salida entonces el resultado será satisfactorio. Obviamente, el algoritmo tiene que permitir una versión paralela. Si el algoritmo necesita el resultado del valor de entrada anterior paralelización será mucho más difícil. Esto se muestra para el algoritmo *factorial*. Este algoritmo es computacionalmente intensivo, pero produce una gran cantidad de datos de salida. Copia de datos desde y hacia la GPU es una tarea que consume mucho tiempo. Cuando se requiere una gran cantidad de datos de entrada o se genera una gran cantidad de datos de salida, la copia de estos datos podría tener un impacto tan grande que no hay más ganancia en tiempo de ejecución cuando se utiliza CUDA. Este trabajo demuestra que ciertos algoritmos que manejan grandes volúmenes de datos se pueden trasladar a CUDA con una disminución en el tiempo de ejecución.

Palabras clave — Hadoop, CUDA, JCuda, Java, C

Glossary

blockDim The dimensions of the blocks on a CUDA device. 7

blockIdx This is a one, two or three dimensional index used to identify the block within the grid of cores on a CUDA device. 7

device The device is refers to the CUDA capable device which is being called, this is the GPU. 13, 17

host The host is the device calling CUDA, this is the CPU. 13

labomat36 The compute cluster from the Escuela Politécnica Superior of the UAM used to test all implementations of all algorithms. 15, 16

stdin Standard input stream. 6, 12

stdout Standard output stream. 6, 12

threadIdx This is a one, two or three dimensional index used to identify the thread within its block on a CUDA device. 7

Acronyms

API Application Program Interface. 8

CPU Central Processing Unit. 1, 6, 7, 9, 13, 17, 21

CUDA Compute Unified Device Architecture. 1–3, 5–8, 11–13, 15–18, 20, 23–26, 29, 30

GPU Graphical Processing Unit. 1, 2, 6–9, 12, 15–17, 20, 21, 29, 30

HDFS Hadoop Distributed File System. 5, 6, 17

JNI Java Native Interface. 1, 7, 12

JVM Java Virtual Machine. 8

NVCC Nvidia CUDA Compiler. 16

OS Operating System. 9

PTX Parallel Thread Execution. 16, 17

Contents

List of Tables

List of Figures

1

Introduction

With the rise of multi-core Central Processing Unit (CPU)'s programmers needed schooling in parallel computing. This knowledge could easily be used for parallel computing in Graphical Processing Unit (GPU)'s. With NVIDIA's newer GPU having hundreds of cores and the development of Compute Unified Device Architecture (CUDA), a C compiler for GPU's, this could mean the next step. With powerful GPU's becoming affordable in the last few years, they are becoming more popular for their parallel computing capabilities. Originally developed to process graphical images they are now also being used to perform other tasks as well. However the storage power and even the compute power of CUDA on a single node has already become a bottleneck.

With both Hadoop and CUDA as leading technologies in their respective fields of big data and programming massively parallel processors, it was only a matter of time to combine the two technologies and get the best of both worlds. This is the goal of this work. Other studies have been previously conducted on this matter. The Hadoop wiki page even proposes a few ways of integrating CUDA in Hadoop. They propose Java Native Interface (JNI) for Java and a *MapRed* class for C/C++. Most of the existing Hadoop-GPU solutions were made for a specific reason and will therefore not apply well

for other problems. The goal here is to provide the reader with a generic solution which can be applied to every problem assuming that the problem fits the profile for CUDA. [1]

1.1 Scope

The Goal of this project is not to deliver a real world application but rather to show the improvements in performance which can be achieved by porting the computationally expensive tasks to CUDA on a cluster running Hadoop. Since CUDA does not handle lots of data well, the difference between applications which require a lot of input or output data and those which do not will be shown. This is done by using two different functions. The first one is *factorial n* and the second is *the sum of the first n integers*. These are both very simple algorithms which only require one integer as input. The *factorial n* function will however generate a very large number as output while the *sum of the first n integers* generates a much smaller output. These algorithms will be written in Java and JCuda and both of them will be tested on a single node and on the Hadoop cluster. JCuda are Java bindings for CUDA. [2] In this project JCuda will be used to make calls to the GPU from Java. This way the mapper and reducer can be written in Java and calls to the GPU can be made from there. Some other ways of calling the GPU will be briefly mentioned but not discussed in depth.

Trying to improve any of the used tools is not part of this project. This means efficiency of Hadoop, JCuda or CUDA will not extensively discussed, nor will trying to improve any of those on its own be part of the project.

1.2 Structure of the document

First all the used technologies and how they work will be briefly explained. This allows the reader better understanding of the work. These technologies include Hadoop, CUDA and JCuda. Next how to tackle the presented problem will be laid out. This includes a short introduction to the possible ways of solving the problem as well as a detailed explanation of the method chosen for this work. The two different ways of implementing both algorithms will also be discussed here. First the Java implementation will be discussed in depth and later the CUDA implementation along with their limits. Afterwards all the details about the programming phase of the project will be explained. This means

in in depth explanation on setting up the cluster, compiling and running a program, difficulties encountered while programming and how to handle unexpected events. It will also explain in detail how the algorithms work on CUDA. The results from running both implementations of both algorithms will be given and interpreted. Finding an explanation for the results will also be done here. The last chapter is the conclusion of the work and what this newly gained information can be used for in the future.

2

State of the Art

This project is build using the following technologies, it uses the Hadoop to divide the workload among the different nodes of the compute cluster. These nodes will use CUDA to perform their most computational expensive tasks. Since the mapper and reducer will be written in Java, JCuda is needed as an interface link between Java and CUDA.

2.1 Hadoop

Hadoop is a framework developed by the Apache Software Foundation to process very large datasets on a compute cluster. The Hadoop Distributed File System (HDFS) is a distributed file system that makes all files in the HDFS available to all nodes. This is where the input and output files for the task will be stored. The input file will be split into FileSplits. These splits happen with no regard to internal structure of the document. The input files will be split if they are too large. One mapper method is launched per FileSplit on a single node. This is a user defined function used to process the received data and is the same for every node. The mapper function is a function which will map every

value from the input to a key-value pair. These key-value pairs are written in temporary files on the HDFS. The reducer will merge sort these temporary files on the key. This way all the key value pairs with the same key are grouped together in a new file. the reducer can now read this file sequentially and all the values with the same key are passed to the same reducer function. This function is also user defined. This leaves the user with one value for each key. These new key value pairs without duplicates in the key are written to the output file on the HDFS. One output file is created per reducer. When the mapper outputs a key-value pair, this pair is already stored in local memory. For efficiency it might be a good idea to combine all the values which have the same key before writing them to a file. This can be done with a combiner. The combiner collects the key-value pairs from the mapper in a list of values for each key before they are written to a file. When a certain amount of pairs have been outputted from the mapper these lists will be passed to the combiner method. The combiner is a user defined function which will write the key-value pairs to the file system with only one value per key. The combiner can be seen as a reducer which only operates on a small part of the key value pairs. The process of calling the mapper followed by the reducer is called MapReduce. Generally all Hadoop functions are written in Java although other possibilities exist. Streaming lets the user write the mapper or reducer functions in any other language. These functions must read their input line by line from stdin and write the output to stdout. Besides streaming Hadoop provides a *MapRed* class for C/C++. [3]

2.2 CUDA

CUDA is a parallel computing platform and programming model invented by NVIDIA. It enables dramatic increases in computing performance by harnessing the power of the GPU. [4] CUDA uses the GPU to perform computational intensive tasks in parallel. This can be done in two ways. The first way is to divide a single task in n smaller subtasks which can then be executed in parallel. From the results of these n subtasks the final result can be calculated in a small amount of time. For the other way it has to be necessary to perform the same task n times. These n tasks can then also be executed in parallel. The difference lies in the fact that in the first way a single result is calculated in parallel while for the second way n results are calculated in parallel. The setup for both approaches is the same and is visualized in Figure 2.1 [5]. The input data is copied from the CPU to the GPU (1). This is a time consuming process and will play an important

role later. Once the data is copied to the GPU the kernel will be run (2). The kernel is the collection of functions to be run on the GPU. Every GPU can only run one kernel at the same time. The kernel will launch n threads (3). These threads are grouped into blocks. If there are not enough CUDA cores available to grant every thread a core the blocks will be sequentially scheduled. Every thread executes the same code. The only way for a thread to differentiate itself from others is by using its threadIdx and blockIdx. By using these two variables, which are available in every thread, a unique global index can ba calculated using the blockDim. Using this global index every thread can identify their part of the input data. Once all threads have successfully finished, every bit of the input data is processed. When the kernel has finished, the result must be copied from the memory on the GPU to the CPU (4). This is once again a time consuming process. It is clear that CUDA excels at tasks which are computationally intensive without using a lot of data. CUDA kernels, their setup and calls to these kernels are written in C. [6]

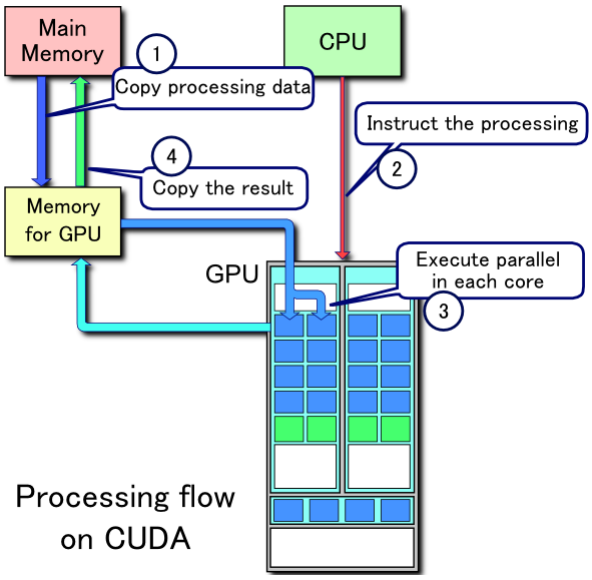

Figure 2.1: Processing flow on CUDA

2.3 JCuda

Since Java will be used to write the mapper and reducer it is necessary to be able to call CUDA from Java. There are multiple possibilities available. It is possible to use the JNI to call C code. This C code can contain the setup for CUDA and the calls to the kernel. An other possibility is JCuda. JCuda are Java bindings for CUDA. These Java

bindings for CUDA will be used to make calls to CUDA from Java. JCuda is written with ease of use in mind. Therefore the JCuda Application Program Interface (API) is kept as close to the original CUDA API as possible. The semantics and signatures are kept consistent with those of CUDA where possible. When using JCuda you first need to initialize the CUDA device and load a kernel. This kernel is still written in C and has to be compiled with the CUDA compiler. The compilation can be done by JCuda during runtime or beforehand by the user. Once the kernel has been loaded a pointer to this function has to be created. This pointer will be used to call the proper kernel. From here on the execution is similar to normal CUDA. The memory on the CUDA device has to be allocated and the input has to be copied to the device. The kernel can now be called. When the kernel finishes its run the output has to be copied back to the host device. JCuda depends on native libraries as well as JCuda jar files. In order for JCuda to work, the Java Virtual Machine (JVM) needs to be able to access the JCuda jar files and the JCuda native libraries. It is therefore important that these libraries are located in a place visible for the JVM. [2]

2.4 Hardware

The cluster used for this project belongs to the *Univisidad Autonoma de Madrid* and is used for the *Arquitecture de Sistemas Paralelos* course. The cluster contains twelve slave nodes and one master node. Table 2.1 shows the hardware of the cluster. There are 4 nodes with a GPU, but one of them is down so only three will be used for this project. The other 8 nodes will not be used at all. The slave nodes used for this project have two GPU's. The Hadoop cluster is defined to allow two map or reduce tasks concurrently on the nodes. This way both of the GPU's can be used by two different mappers or reducers. Increasing performance by adding hardware is not a part of this work.

	CPU	Operating System (OS)	GPU
1 Master Node	AMD(R) Opteron(TM) Processor 6128 @ 800MHz	CentOS release 6.3	
4 Slave Nodes (used)	Intel(R) Core(TM)2 Quad CPU Q9450 @ 2.66GHz	CentOS release 6.3	2 times GeForce GTX 480
8 Slave Nodes (unused)	Intel(R) Core(TM)2 Quad CPU Q9450 @ 2.66GHz	CentOS release 6.3	

Table 2.1: Hardware of the cluster

3

Design

In order to show the difference between Hadoop and Hadoop combined with CUDA every algorithm needs to be implemented twice. A normal Java version and a version using JCuda and CUDA are required. Using these two implementations of the same algorithm a graph can be made showing the time complexity of both methods. Since CUDA is only effective for algorithms which meet some specific conditions it is important to show the difference between an algorithm that does satisfy those conditions and one that does not satisfy those conditions. The algorithms which will be used are very similar in nature but there is a difference in the amount of output they generate. The algorithms are *sum of first n integers* and *factorial of n*. Both algorithms only need a single integer as input and return a single value. The difference lies in the size of the output value. Because this project is about big data these algorithms will be used to calculate the result of the corresponding function for millions of values for n in a single run. These values will be stored in a file. This file will be used as the input file for Hadoop. Different input files will be used for both algorithms because the *factorial n* generates a much larger value for every input value which will have to be stored in multiple integers. The input values for *factorial n* will therefore be kept smaller than those of the *sum of first n integers*. However the same

input files will be used for the two different implementations of the same algorithm.The CUDA implementations will be kept as close to the Java implementations as possible. This is to make sure the difference in execution time is not implementation dependent. Hadoop supplies the mapper with a string of input values. This string has to be tokenized. Once tokenized every value will be gotten and processed sequentially.

JNI can be used to call C code from Java. When the kernel and the calls to this kernel are written in C the JNI can be used to execute this C code and run the kernel from C instead of Java. Another way of calling the kernel is by using Hadoop Streaming. This allows the user to execute an external program which will read its input from stdin and write the output to stdout. This external program can be a C program with a CUDA kernel. This program will read stdin and store these values in a list. This list can then be copied to the GPU for processing by the kernel. The output list can then be written to stdout in a loop.

3.1 Java

The Java implementation will loop over all the tokens and call the algorithm once for every token and write the output to the output collector. This means that the algorithm will be called as many times as there are values in the input file. Every mapper will only get a part of the input file and will process its part completely sequentially. The *sum of first n integers* algorithm will iterate over all positive natural numbers smaller or equal to the token and add them to a total sum. The closed formula $(n^2 + n)/2$ will not be used because the task has to be computationally intensive. It is clear that the *sum of first n integers* function has a quadratic growth relation between the input and output values. The factorial function will be calculated in the same way but instead of adding the value to the result it will now be multiplied with it. The growth factor of the factorial function is larger than that of the exponential function. This means that the result of the factorial function gets really large. Due to this growth the result will almost never fit in a single integer therefore it will be stored in multiple integers.

3.2 CUDA

When using CUDA, the algorithm will not be called immediately when a token is received. Instead this token will be added to a list. When this list has grown sufficiently large the algorithm written in JCuda and CUDA will be called. Once again this works the same for both algorithms. JCuda will first be used to set up the device. Once this is done the list containing the values will be copied to the device and memory for the result will be allocated. The kernel will launch a thread for every value in the list. If there are not enough threads available simultaneously, the blocks will be scheduled sequentially. These threads execute the exact same algorithm as the CPU does in the Java implementation. Every thread will store his result in the memory allocated for the results. Once all threads are finished the results will be copied from the device to the host. The values in the result list will be written to the output collector in a loop.

Clearly there is a lot of overhead. First of all not all input values are available from the start since they are stored in a string which has to be iterated over to convert all input values to integers and store them in the input list. Once the kernel finishes its task, the output has to be copied to the host. Afterwards all the values in the output list have to be sent to the output collector. This will also be done in a loop. This means that in order for the CUDA implementation to provide better performance the overhead needed to actually execute the CUDA version of the algorithm has to be taken into account. The call to the kernel however is non blocking. This means that while the kernel is running the CPU can perform other tasks like writing to the output collector or creating a new input list.

4

Development

4.1 Setting up the cluster

In order to be able to run any Hadoop job which uses JCuda as bindings to call CUDA, the labomat36 cluster first has to be set up properly. The labomat36 has to be set up for Hadoop, CUDA and JCuda. The labomat36 cluster consists of twelve slave nodes. Four of those nodes have a GPU and the others do not but since one node with GPU was down, only three nodes were used. The master node used to access the cluster also does not have a GPU.

For Hadoop there was opted to go with Cloudera CDH. CDH is the worlds most complete, tested, and popular distribution of Apache Hadoop and related projects. CDH is 100% Apache-licensed open source and is the only Hadoop solution to offer unified batch processing, interactive SQL, and interactive search, and role-based access controls. [7] This is due to its ease of use. CDH can easily be installed and maintained on a cluster using the Cloudera Manager. This is a tool used to install and upgrade CDH using a web interface. The installed version of Hadoop is "Hadoop 2.5.0-cdh5.3.2" which means it is

version 5.3.2 of CDH running Hadoop version 2.5.0. CDH is set up with three compute nodes, all of which have a CUDA capable GPU, and one master node. The master node will be used to issue jobs to the cluster but will not be used for mapping or reducing. The nodes which do not have a GPU will not be used by CDH.

CUDA needs to be installed on all the compute nodes in the cluster. The installed version on labomat36 is "CUDA Toolkit version 4.1". CUDA can usually be installed from the official repositories. If CUDA is not available in the official repositories it can be downloaded from NVIDIA's website. JCuda is needed to call CUDA from Java and also needs to be installed on all the compute nodes. The installation of JCuda consists of two parts. The first part are the native libraries. JCuda relies on native libraries which will be called at runtime. This means that these libraries need to be placed in a location visible at runtime. Since the execution will be done using Hadoop they need to be placed in a location visible for Hadoop. Hadoop will look for libraries in its "lib" folder. With the CDH version on the cluster the full path to that folder is "/opt/cloudera/parcels/CDH-5.3.2-1.cdh5.3.2.p0.10/lib/hadoop/lib". Once all the required native libraries are in that location the first step is complete. JCuda supplies native libraries which are not required for the core functionality. The next step are the Java libraries. JCuda uses .jar files which have to be included in the source code of the project in order to have access to the functionality JCuda provides. When the JCuda jars are included they also have to be visible at runtime. This is solved by packing the .class files of the JCuda jars in the jar of the application.

4.2 Compiling and running a program

A certain directory structure will be used to keep track of all files. This structure requires the "./src", "./kernel", "./bin" and "./classes" folders in the root folder of the project. The "./src" folder will contain all source code. This includes the .java and .cu (CUDA) source files. The compiled kernel will be located in "./kernel". Once the .java files are compiled to .class files these will be stored in "./classes", this folder will also contain the JCuda .class files. The final jar will be placed in "./bin". Compiling, packing and running a program can be done by using the commands in Code 4.1. The .cu files which contain the kernel code have to be compiled to Parallel Thread Execution (PTX) using Nvidia CUDA Compiler (NVCC). A PTX file is a file containing pseudo-assembly code used in the CUDA environment. This file will be loaded to the GPU which compiles it to binary

code which can be run on the cores of the GPU. The compiled kernel has to be placed in the HDFS in order to load it in the map or reduce phase.

Code 4.1: Compile and package program into jar

```
# Compile the .java files
javac -classpath <location to hadoop-core-version.jar>:jcuda/jcuda.
   ↪ jar -d classes src/<source>.java
cd classes/
# Pack the .class files in a .jar
jar -cf ../bin/<name>.jar *.class jcuda/*.class jcuda/*/*.class
# Run the program with Hadoop
hadoop jar bin/<name>.jar <main class> <path to input files> <path
   ↪ to output files>
```

4.3 Writing the program

This section will not go in depth about how to run a Hadoop job or a JCuda job. Information about this can be found on their respective documentation websites. What will be discussed is how to set everything up to use the GPU from Hadoop. First the .ptx file containing the kernel has to be added to the distributed cache. This makes sure the file is available to all nodes. This has to be done in the body of the main function as this only has to be done once. Next step is to set up JCuda as described in their documentation for the mapper or reducer. Both the algorithms used in this work will only use the mapper. The reduce phase will be entirely skipped. When loading the module the kernel compiled to PTX has to be gotten from the distributed cache. The next step is to set the CUDA device up from JCuda using the code in 4.2. When the JCuda device is properly set up it is time to allocate memory for the input and output of the kernel. This will be further explained for each algorithm in their respective sections. When all memory is allocated and the input is copied to the device, the kernel can be called. While the kernel is running the output from the previous kernel can be written to the output collector and the new input can be prepared. Take note of two special cases. The first time the kernel is called there will not yet be any output to write. The last time the kernel will be called no new input has to be prepared. This allows the CPU and GPU to work in parallel and will increase performance.

Code 4.2: Setup JCuda

```
// Initialize the driver and create a context for the first device.
cuInit(0);
CUdevice device = new CUdevice();
cuDeviceGet(device, 0);
CUcontext context = new CUcontext();
cuCtxCreate(context, 0, device);

// Load the ptx file.
CUmodule module = new CUmodule();
cuModuleLoad(module, localFiles[0].toString());

// Obtain a function pointer to the kernel function.
CUfunction function = new CUfunction();
cuModuleGetFunction(function, module, "factorial");

// Allocate the device input and output data
CUdeviceptr deviceInput = new CUdeviceptr();
cuMemAlloc(deviceInput, <size of input>);

CUdeviceptr deviceOutput = new CUdeviceptr();
cuMemAlloc(deviceOutput, <size of output>);


// Copy hostInput to the device
cuMemcpyHtoD(deviceInput, Pointer.to(hostInput), <size of input>);
```

4.3.1 Sum of first N integers

When using Java the sum of all input values will be calculated and written to the output collector before getting the new input value. This means that all calculations and data management will happen sequentially. This is how Hadoop is normally used. Since the algorithm is simple and no external functionality apart from Hadoop is used this implementation will not be discussed in much more detail. It will however be used to gain results which can be compared to the results of the CUDA implementation of the algorithm.

The CUDA implementation will receive a pointer to a list of appropriate length to

store all the results. The second argument is a pointer to the list of input values. the last argument is the length of the input list. As seen in Code 4.3 the partial sums will be stored in shared memory for faster memory access. When the computations are done the result is copied to the output list. Because every thread calculates a single sum, as many sums as there can be threads at the same time can be calculated.

Code 4.3: Sum of First N Numbers in CUDA

```
extern "C"
__global__ void sum_of_first(long* output, unsigned int* input, int
    ↪ n){
  int index = threadIdx.x + blockDim.x * blockIdx.x;
  extern __shared__ long result[];
  if(index < n){
    result[threadIdx.x] = 0L;
    for (unsigned int i = 1; i <= input[index]; i++){
      result[threadIdx.x] += i;
    }
    output[index] = result[threadIdx.x];
  }
}
```

4.3.2 Factorial N

The Java implementation of factorial n is not as straight forward as that of the sum of the first n integers. The result of the factorial function has a growth factor greater than quadratic growth. This means the results will get very large very fast and will therefore usually not fit in a single integer. To solve this problem the result will be stored in multiple longs. The longs together will hold the result. Storing the result this way implies that the result is a list of longs. The way the result is calculated is the following, the result is stored backwards. The least significant long is stored first and the most significant long is stored last. Let a be the current result which is a list of longs and let b be an integer. Multiplying a with b will be done in a loop which loops over all the longs in a. Let c be the current long in a. Multiply c with b and add the overflow of the previous iteration. This value modulo 1000000000 (this number is obtained by getting the maximum amount of digits in a long minus one, the one will be used as overflow digit) will become the new value for c and the result of the division from this number by 100000000 will be the

overflow used in the next iteration.

This same algorithm can be used in CUDA. Again the task of calculating one factorial will not be done in parallel but instead multiple factorials will be calculated at the same time. As can be seen in Code 4.4. This implementation requires to define the size of the result beforehand. When the size of the result is not sufficient to store the entire result it will only store the last part of the result. This part however will be correct. When dealing with very large input values the results will also grow drastically. This translates to a lot a required memory for the result. This is one of the weak points in CUDA.

Code 4.4: Factorial of N in CUDA

```
extern "C"
__global__ void factorial(long* output, unsigned int* input, int n,
   ↪  int length){
  int index = threadIdx.x + blockDim.x * blockIdx.x;
  if(index < n){
    for (unsigned int i = 1; i < length; i++){
      output[index * length + i] = 0L;
    }
    output[index * length] = 1L;
    for (unsigned int i = 1; i <= input[index]; i++){
      long x;
      int ll = 0;
      for (unsigned int j = 0; j < length; j++){
        x = output[index * length  + j] * i + ll;
        output[index * length + j] = x % 1000000000;
        ll = x / 1000000000;
      }
    }
  }
}
```

4.4 Testing the code

A test setup was used in order to test all the algorithms and implementations on a personal laptop running Arch Linux and a GeForce GT 560M GPU. On this laptop a single node Hadoop cluster was installed. The version however differentiates from the version on the

labomat36 cluster. The Hadoop version of the single node cluster is "Hadoop version 1.2.1". The version on the labomat36 is fully backwards compatible with the version on the single node cluster. There is a difference in the location of the native libraries between the two versions but this is due to the fact that Hadoop was installed on the labomat36 using Cloudera and on the laptop using the official Arch repositories. Once the code was tested on the single node cluster, it could be moved to the labomat36 to get the results.

4.5 Exceptions

Hadoop takes care of load balancing and distributing the tasks among the slave nodes. Since Hadoop is designed to be robust not a lot can go wrong here. The only problem which can arise is a node without a GPU. When a one or more nodes do not have GPU's the kernel cannot be executed on those nodes. This will throw an exception from JCuda saying it failed to load one of the native JCuda libraries. It is therefore important not to have the JCuda native libraries visible on the nodes without a GPU. The exception thrown cannot be caught by a try catch block. Since it is being handled by JCuda. The way to make a program robust for this kind of errors is by creating a boolean variable which will maintain whether or not there is a GPU. By default this variable is set to false. Once the setup of JCuda completes successfully it is set to true. The setup of JCuda and the calls to the kernel are done in a try catch block. This block however does not have a catch statement but it has a finally block. In this block the boolean can be tested. When it is still false after the setup this means an error occurred during setup and there is no GPU available. Instead of executing the algorithm on the GPU it can now be executed on the CPU. Errors during runtime of the kernel can be detected in the same way. This way the Java implementation can be used as a fall-back when the GPU fails.

5 Results

The input files for both implementations of the two algorithms will be randomly generated. The amount of values will vary as well as the range between which the value must be. By doing this the execution time can be viewed in function of the size of n and in function of the amount of values for n. For the execution time in function of the size of n 1 million input values will be used for both algorithms. For the execution time in function of the size of the input file the input values will vary between 0 and 1000000 for the sum of the first n integers while they will vary between 0 and 1000 for the factorial n. The constants are high enough to for CUDA to outperform Java.

5.1 Sum of first n integers

5.1.1 Execution time in function of the size of n

As can be deducted from Figure 5.1 there clearly is a linear correlation between the size of n and the execution time for both Java and CUDA when the value of n becomes large

enough. This behaviour was expected and can be explained by the fact that when the problem becomes x times bigger, the execution time will also become x times bigger. The execution time for both implementations rises equally fast. For low values of n the execution time remains constant. This is due to the overhead created by Hadoop. Hadoop has to set the task up and divide the input over the compute nodes. The graph also shows that for these low values of n the execution time for CUDA is actually longer than for Java. The cause of this is once again set up time. CUDA has to set up the device and copy the input to it. This set up time is constant no matter the size of n. With higher values for n the set up time can be neglected in comparison to the time actually spent computing the results. For the larger values of n the execution time of Java overtakes the execution time of CUDA. This also is expected behaviour because computation time becomes a large enough factor to make up for the set up time of CUDA. For those values of n CUDA is about eight times faster than the Java implementation. This is a noteworthy increase in performance. The CUDA implementation does not get better by increasing the size of n, meaning that the factor by which it is faster will remain the same.

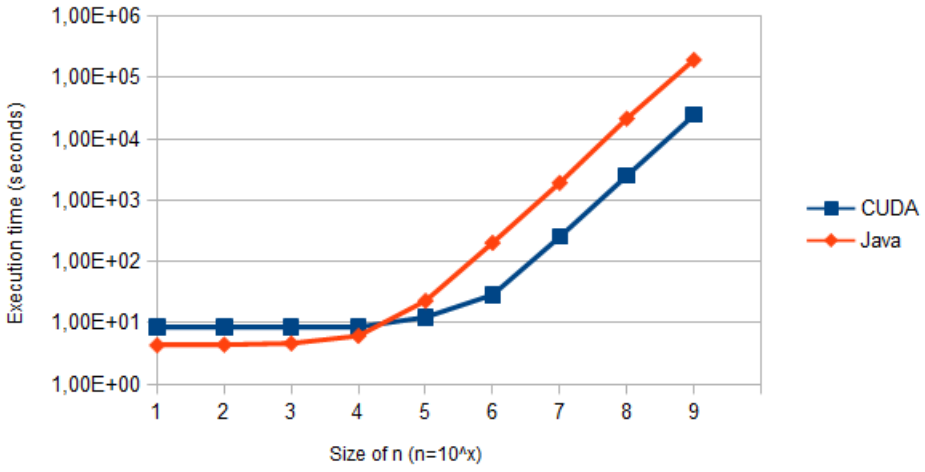

Figure 5.1: Time in function of the size of n for the sum of first n integers

5.1.2 Execution time in function of the size of the input file

Figure 5.2 clearly shows a linear correlation between the amount of input values and the execution time of both the CUDA and Java implementations. This was also to be expected. Increasing the amount of input values by a factor x means x more calculations have to be done. For the Java implementation it is clear this will take x times longer. For

CUDA the same applies once the initial amount of input values is high enough. When the initial amount fills up an entire list which will be passed to the kernel, increasing the amount of input files by a factor x means that the amount of input lists will also increase by a factor x. This means the kernel has to be called x more times. When the initial amount is not high enough the execution time will remain almost the same because no extra kernels have to be launched. The output still has to be written to the output collector which will take some time. The execution time for small amounts of input values are constant. This is once again due to the overhead created by Hadoop and in the case of CUDA to the set up of CUDA. Increasing the amount of problems does once again not increase the factor by which the CUDA implementation is better.

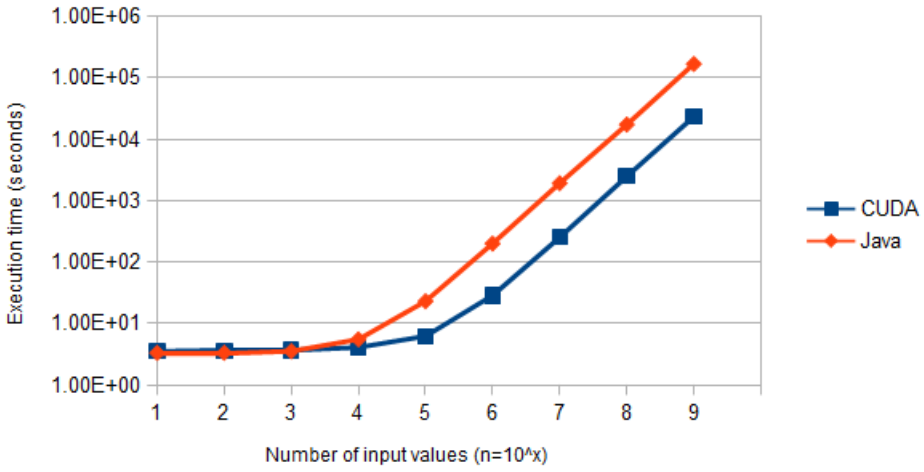

Figure 5.2: Time in function of the amount of input values for the sum of first n integers

5.2 Factorial n

5.2.1 Execution time in function of the size of n

For the factorial function an almost linear correlation between the execution time and the size of n can be observed from Figure 5.3. The slope however is steeper than the slope of the sum of first n integers and increases slightly for larger values of x. This is caused by the increase in size of the result. This can be explained by the way the result is stored. For larger values of n the result becomes to big to be stored in a single long. This is why a list of longs is used. When the size of the problem is increased by x, x times more

multiplications will have to be done on *increase in size of result* more longs. In the design chapter the growth of the result has already been discussed. For small values of n Java is once again clearly the best option. The reason for this is the same as in the case of the sum of first n integers. CUDA however catches up very quickly with Java, at which point their execution times are very close together. Porting this algorithm to CUDA is therefore not a good idea. There is no increase in performance, there is actually a small decrease, and CUDA uses more resources than Java.

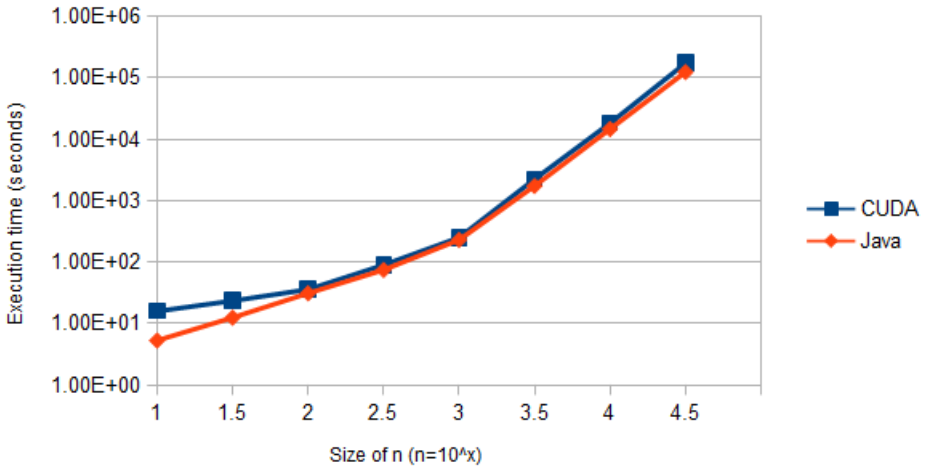

Figure 5.3: Time in function of the size of n for factorial n

5.2.2 Execution time in function of the size of the input file

The same linear correlation as with the execution time in function of the size of the input for the sum can be seen in Figure 5.4. This cause for this is the same as before. Increasing the amount of problems by a factor x means translates to an increase in execution time of factor x. The exponential growth from the execution time in function of the size of n does not recur. Adding more problems does not have the same impact on the size of the result as increasing the value for n does. For small amounts of input values there is once again a constant execution time due to the overhead.

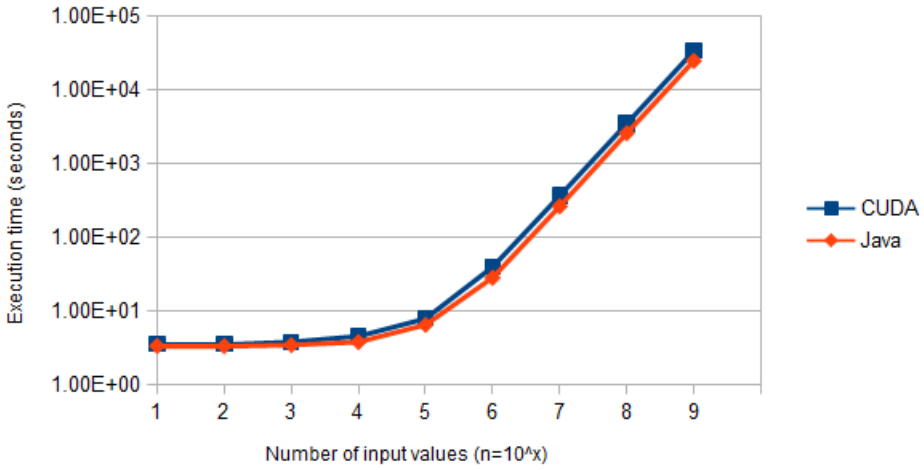

Figure 5.4: Time in function of the amount of input values for factorial n

6

Conclusions

It is clear that when the problem allows for a CUDA implementation of its computationally intensive parts, improvements in performance can be achieved. This does not mean that porting every problem using Hadoop to Hadoop combined with CUDA is a good idea. As shown in the previous chapter the factorial function is not a viable option to port to CUDA because there is too much overhead in copying data. Other examples of bad algorithms are those which do not take any time to compute. Computing the sum of the first n integers using the closed formula is one such example. The time calculating the sum does not justify the use of a GPU. In other words any algorithm that might be ported to CUDA in a normal environment would most likely be a good option to port to CUDA in the Hadoop environment. Another downside is the cost. The High-End family of GPU's designed for computing like the Tesla are very expensive. These GPU's are made to run 24/7. The Gaming family like the GeForce however can give good performance are reasonable prices. No matter the GPU, upgrading an entire cluster remains an expensive investment which has to pay out in the long run.

Combining Hadoop with CUDA can be a useful solution for time sensitive tasks or maybe even tasks which have to be performed in real time. When time is not an important

factor it might be better to stay with the traditional Hadoop without CUDA due to the price tag of a cluster equipped with GPU's.

Bibliography

[1] *CUDA Performance Study on Hadoop MapReduce Clusters CSE 930 Advanced Computer Architecture*, University of Nebraska-Lincoln Std.

[2] *JCuda*, Std. [Online]. Available: http://www.jcuda.org/

[3] *wiki.apache.org/hadoop*, Apache Std. [Online]. Available: http://wiki.apache.org/hadoop

[4] *CUDA*, NVIDIA Corporation Std. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[5] *Wikipedia CUDA*, Std. [Online]. Available: https://en.wikipedia.org/wiki/CUDA

[6] *Programming Massively Parallel Processors: A Hands-on Approach*, Std., 2012. [Online]. Available: https://books.google.com/books?id=E0Uaag8qicUC&printsec=frontcover&hl=en&source=gbs_ge_summary_r&cad=0

[7] *Cloudera*, Std. [Online]. Available: http://www.cloudera.com/content/cloudera/en/products-and-services.html

Appendix